# 4

# THINKING ABOUT PROTOTYPING

NOW THAT WE'VE looked at the principles of design and the fundamentals of Internet communications, we hope you are itching to create an Internet of Things device! It's possible that you want a single device that is Just For You. But perhaps you have a fantastic idea and are planning to churn out millions of the products. In both cases, the most sensible approach is to start by making one Thing first: a prototype.

Making a prototype first has many benefits. You will inevitably come across problems in your design that you need to change and iterate. Doing this with a single object is trivial compared to modifying hundreds or thousands of products. With the Internet of Things, we are always looking at building three things in parallel: the physical Thing; the electronics to make the Thing smart; and the Internet service that we'll connect to. The last of these is relatively cheap and easy to change. You cannot change the physical object and its silicon controller unless you recall every item.

The prototype, therefore, is optimized for ease and speed of development and also the ability to change and modify it. Many Internet of Things projects start with a prototyping microcontroller, connected by wires to components on a prototyping board, such as a "breadboard", and housed in some kind of container (perhaps an old tin or a laser-cut box). This prototype is relatively inexpensive, but you will most likely end up with something that is serviceable rather than polished and that will cost more than someone would be willing to pay for it in a shop.

At the end of this stage, you'll have an object that works. It may be useful for you already. It may be a talking point to show your friends. And if you are planning to move to production, it's a *demonstrable* product that you can use to convince yourself, your business partners, and your investors that your idea has legs and is worth trying to sell.

Finally, the process of manufacture will iron out issues of scaling up and polish. You might substitute prototyping microcontrollers and wires with smaller chips on a printed circuit board (PCB), and pieces improvised out of 3D-printed plastic with ones commercially injection-moulded in their thousands. The final product will be cheaper per unit and more professional, but will be much more expensive to change.

In the next few chapters, we look at aspects of prototyping in detail:

- Chapter 5: Choosing a specific microcontroller for prototyping
- Chapter 6: Designing a housing, any moving parts, and so on for the physical object that is Good Enough for the prototype
- Chapter 7: Developing the web service that the device will speak to

In this chapter, we discuss some more theoretical issues about the approach to choosing a prototyping platform. We could easily give a pat answer like "Use an Arduino and develop your online service in Ruby on Rails" or "Use a Raspberry Pi", but determining the best choice of prototyping platform can't come from on high or from a book. For one thing, your project has its own goals and requirements. For another, new microcontrollers are constantly coming onto the market; best practice and popularity of server software stacks is ever changing; and rapid prototyping continues to evolve—so we feel it is worth giving some more general advice and background information before digging deeper.

## SKETCHING

There is a good chance that the first step you'll take when working on your prototype will be to jot down some ideas or draw out some design ideas with

pen and paper. That is an important first step in exploring your idea and one we'd like to extend beyond the strict definition to also include sketching in hardware and software.

What we mean by that is the process of exploring the problem space: iterating through different approaches and ideas to work out what works and what doesn't. The focus isn't on fidelity of the prototype but rather on the ease and speed with which you can try things out.

For the physical design, that could mean digging out your childhood LEGO collection to prototype the mix of cogs and three-dimensional forms, or maybe attacking some foamcore or cardboard with a craft knife. We examine such techniques in more detail in Chapter 6, "Prototyping the Physical Design".

To show how you might approach "sketching" for the electronics and software, an example will help.

The Internet of Things design firm BERG invited me (Adrian) along to their inaugural Little Printer hackday in June 2012. They filled their office with a bunch of interesting techies and creatives and tasked them with seeing what they could do, in a day, with BERG's (at the time) soon-to-be-released cute Internet-connected diminutive printer. (For more on the Little Printer, see the case study in Chapter 10, "Moving to Manufacture".)

Most of the attendees focused on creating new publications for the Little Printer—a task that meant writing server code to wrangle data (working with the Google Calendar API, spotting meteors passing overhead, and so on) into shape and experimenting with ways of displaying that on a narrow strip of receipt paper. I (Adrian) decided that having a connected device as the output for the system wasn't enough and spent the day prototyping a custom-hardware input device, too. Called the *Printernet Fridge*, it was a nod to the age-old Internet of Things cliché, the Internet fridge—an exercise in seeing what a semi-automated shopping list would be like.

From the design constraints (mostly how the Little Printer publishing system works but also limited by the hardware that I had thrown into my bag to take to the event), it was clear early on that the problem could be broken into three broad areas: the graphic design of the printed publication; the physical hardware to easily add items to the shopping list; and some server software to tie the rest of the system together. Breaking the problem into these three parts meant that, initially at least, each could be addressed separately.

The first step was to pull together the scaffolding of the server software, which other parts of the system could be built on as they were developed.

As this was a prototype, rather than a system to be deployed to thousands of users, I used the simple framework Sinatra. Sinatra is a way to quickly build web services, much like the Dancer framework which we cover in Chapter 7, "Prototyping Online Components", but using the Ruby programming language.

Given that the prototype was going to be demonstrated only to a handful of people and didn't need to extend beyond a single user, I didn't waste any time including a login system for multiple users or setting up any security to encrypt the requests to and from the service. That provided enough infrastructure to interact with the Little Printer publication system and to allow the addition of API hooks for the input device to call when it was ready. I created a crude placeholder image for the publication's icon and dashed off the description text with a simple "prints shopping lists", to let the focus stay on the server software rather than get sidelined into the design and polish. Similarly, the publication itself was just the bare minimum static text which was delivered whenever the publication was requested.

At this point it was possible to subscribe to the publication and have it print a set, one-line document on a Little Printer. Collaborating with one of the designers at the hackday, we iterated through the look and layout of the shopping list by tweaking the HTML, CSS, and images in the Sinatra app that made up the publication.

The content was still static—you would always get a list asking for two bottles of milk and some fresh orange juice—but the header image and text were refined and decisions made about how to convey the information to the user. The simple bullets for each list item were replaced with numbers of each item required, which then moved to the end of the list item so the bullet could be reinstated but as an empty check box to allow you to tick items off as you found them in the shop.

All it needed now was some live data. The workflow for that needed to be as easy as possible; it shouldn't be a chore to add items. However, this wasn't to be some seamless vision of the future where the fridge would order things for you. (The ultimate agency of the user is important.) It is a tool rather than an autonomous decision maker.

With only an Arduino Ethernet board in my bag, the decision over hardware platform was already made. For a production unit, the Ethernet cable would complicate installation, and encrypting the data being sent would stretch the board's limits; however, for a quick prototype, the ease of wiring up different circuits and changing the onboard code easily outweighed the longer-term disadvantages.

Once a basic pushbutton was wired up to the Arduino, I could write some simple code to trigger whenever the button was pressed. The first step merely output some text to the serial port, which could be monitored immediately over the USB cable on my laptop.

Now that the basic physical interaction was working, the next step was to hook it into the web service. That required me to change the software on both the Arduino and the web service. I added a new API call to the Sinatra app which allows new items to be added to the list and tested it with a web browser (as that gives an easier view of what is happening if things go wrong).

For a more complex project, I would have chosen Rails over Sinatra as the web framework. Doing so would have let me pull in the RailsAdmin module to check things like whether the item had been added to the database correctly. Such helper modules let you focus on sketching out one feature, without having to divert to building chunks of additional infrastructure before you have decided that the feature is going to stay. In this case, I could easily switch the static shopping list over to use the live data, which served as both feature development and debugging aid.

Then I pulled in some sample code to make a web request from the Arduino and modified it to call the newly written API on the web service. With that written, it was possible to push a button and have "milk" added to your shopping list. Success!

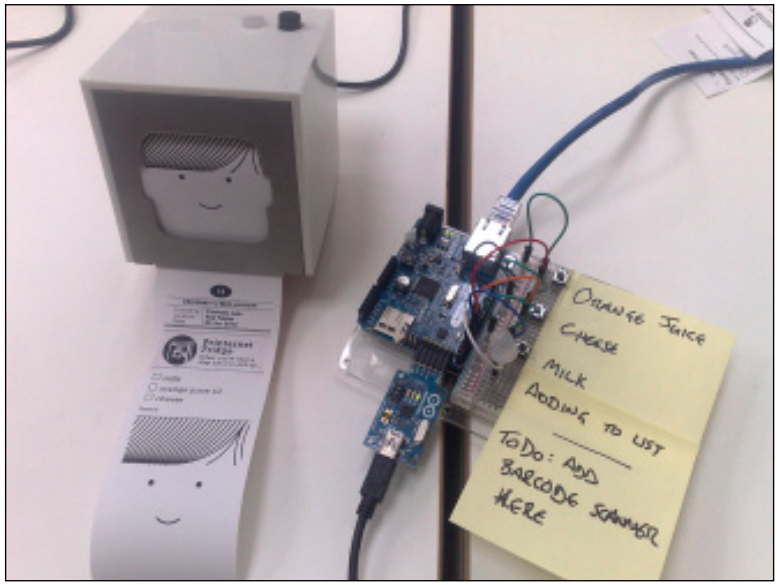
The next step was to add more buttons, as only being able to order more milk isn't the most useful of shopping list applications. Two more buttons were added and connected up to allow "cheese" and "orange juice" to also be ordered.

A quick round of user testing (or showing it off to fellow hackday attendees, as it is also known in this case) highlighted a problem with the design as it stood. Although recording a new selection to the server took less than a second, it was still long enough for subsequent button presses to be missed if you were running through them quickly.

Given that the end of the day was looming large, and as I would be able to talk around the issue in the demonstration, I chose expediency over the larger amount of coding required to decouple the user interface of buttons from the network communication. A "busy" LED was added to the bread-board and illuminated whenever the Arduino was talking to the network.

The work on the hardware and software left little time for developing a case for the input device, which is just as well because I hadn't room for any

construction materials in my bag. Improvising with some sticky notes at least made the interface more self-explanatory. It also allowed room to hint at ways in which the prototype could be further extended, with "Add barcode scanner here" written on a note below the buttons. That was a result of further thinking through how the device might work in practice. You would want a number of buttons, with a web interface to let you reconfigure them for your set of non-packaged fridge goods, and then a barcode reader would allow scanning of anything that was in packaging.




The finished prototype of the Printernet Fridge.

# FAMILIARITY

Another option to consider is familiarity. If you can already program like a whiz in Python, for example, maybe picking a platform such as Raspberry Pi, which lets you write the code in a language you already know, would be better than having to learn Arduino from scratch.

The same applies to the server software, obviously. When creating the Printernet Fridge prototype, Adrian hadn't used Sinatra before but chose it because he was looking for a simple web framework and was already familiar with Ruby from writing a number of Ruby on Rails applications in the past.

And if you're already adept at fashioning sheets of foamcore into three-dimensional structures, we're not going to argue that you should ignore that expertise in favour of learning all about laser cutting.

## COSTS VERSUS EASE OF PROTOTYPING

Although familiarity with a platform may be attractive in terms of ease of prototyping, it is also worth considering the relationship between the costs (of prototyping and mass producing) of a platform against the development effort that the platform demands. This trade-off is not hard and fast, but it is beneficial if you can choose a prototyping platform in a performance/capabilities bracket similar to a final production solution. That way, you will be less likely to encounter any surprises over the cost, or even the wholesale viability of your project, down the line.

For example, the cheapest possible way of creating an electronic device might currently be an AVR microcontroller chip, which you can purchase from a component supplier for about £3. This amount is just for the chip, so you would have to sweat the details of how to connect the pins to other components and how to flash the chip with new code. For many people, this platform would not be viable for an initial prototype.

Stepping upwards to the approximately £20 mark, you could look at an Arduino or similar. It would have exactly the same chip, but it would be laid out on a board with labelled headers to help you wire up components more easily, have a USB port where you could plug in a computer, and have a well-supported IDE to help make programming it easier. But, of course, you are still programming in C++, for reasons of performance and memory.

For more money again, approximately £30, you could look at the BeagleBone, which runs Linux and has enough processing power and RAM to be able to run a high-level programming language: libraries are provided within the concurrent programming toolkit Node.js for JavaScript to manipulate the input/output pins of the board.

If you choose not to use an embedded platform, you could think about using a smartphone instead. Smartphones might cost about £300, and although they are a very different beast, they have many of the same features that make the cheaper platforms attractive: connection to the Internet (usually by wireless or 3G phone connection rather than Ethernet), input capabilities (touchscreen, button presses, camera, rather than electronics components),

and output capabilities (sound, screen display, vibration). You can often program them in a choice of languages of high or low level, from Objective C and Java, to Python or HTML and JavaScript.

Finally, a common or garden PC might be an option for a prototype. These PCs cost from £100 to £1000 and again have a host of Internet connection and I/O possibilities. You can program them in whatever language you already know how to use. Most importantly, you probably already have one lying around.

For the first prototype, the cost is probably not the most important issue: the smartphone or computer options are particularly convenient if you already have one available, at which point they are effectively zero-cost. Although prototyping a "thing" using a piece of general computing equipment might seem like a sideways step, depending on your circumstances, it may be exactly the right thing to do to show whether the concept works and get people interested in the project, to collaborate on it, or to fund it.

At this stage, you can readily argue that doing the easiest thing that could possibly work is entirely sensible. The most powerful platform that you can afford might make sense for now.

Of course, if your device has physical interactions (blowing bubbles, turning a clock's hands, taking input from a dial), you will find that a PC is not optimized for this kind of work. It doesn't expose GPIO pins (although people have previously kludged this using parallel ports). An electronics prototyping board, unsurprisingly, *is* better suited to this kind of work. We come back to combining both of these options shortly.

An important factor to be aware of is that the hardware and programming choices you make will depend on your skill set, which leads us to the obvious criticism of the idea of "ease of prototyping", namely "ease... for *whom*?"

For many beginners to hardware development, the Arduino toolkit is a surprisingly good choice. Yes, the input/output choices are basic and require an ability to follow wiring diagrams and, ideally, a basic knowledge of electronics. Yet the interaction from a programming point of view is essentially simple—writing and reading values to and from the GPIO pins. Yes, the language is C++, which in the early twenty-first century is few people's idea of the best language for beginners. Yet the Arduino toolkit abstracts the calls you make into a `setup()` function and a `loop()` function. Even more importantly, the IDE pushes the compiled code onto the device where it just runs, automatically, until you unplug it. The lack of capabilities of the board presents an advantage in the fact that the interaction with it is also streamlined.

## Case Study: Bubblino

Let's look in some more detail at Bubblino, to see how the process of prototyping played out for a real project.

Bubblino has, from the start, targeted the Arduino hardware, in part because its original purpose was precisely to demonstrate "how to use Arduino to do Internet of Things stuff". So the original hardware connected an Arduino to the motor for an off-the-shelf bubble machine. The original prototype had a Bluetooth-enabled Arduino, which was meant to connect to Adrian's Nokia phone, which was programmed with Python for Series 60. The phone did the hard work of connecting to the Internet and simply sent the Arduino a number, being the number of recent tweets. Bubblino responded by blowing bubbles for that many seconds.

Although Python is a scripting language, with a reputation for being easy to learn, at the time the Series 60 port wasn't so mature and using it turned out more difficult than expected. For the next version, Adrian fell back to using a Perl script on his laptop which worked in exactly the same way, for example, sending a number to the Arduino. This approach worked by opening and writing to a COM port that represented the Bluetooth connection. The same technique could have been used with the current basic USB Arduino. Of course, in both cases, this limits Bubblino to working either within Bluetooth range or with a physical USB connection.

Although it's perfectly reasonable to have a laptop open to drive the device for a demo, Bubblino has now scaled up a little: you can now commission one, which will be hand-built within a few weeks. Although this isn't mass production, the process still requires some streamlining to be cost and time effective. So the current devices are based on an Arduino Ethernet. This means that the Twitter search and XML processing are done on the device, so it can run completely independently of any computer, as long as it has an Ethernet connection.

The original Perl version could use a full-featured XML parser because a wealth of them are available in Perl's collection of libraries, the CPAN (as there will be for any popular modern language). Such libraries do, of course, exist for C++ too. However, due to the memory limitations of the Arduino, loading and processing a large chunk of XML isn't the most appropriate thing to do. Instead, when Bubblino downloads the Atom XML feed for its Twitter search, it simply scans the returned XML looking for the `<published>` tags and the date fields within them. Normally, and with justification, programmers are advised against parsing XML by simple scanning or regular expressions. However, for a limited device, this solution may be acceptable.

In a final twist, the concept of Bubblino has been released as an iPhone app, "Bubblino and Friends", which simply searches Twitter for defined keywords and plays an animation and tune based either on the original Bubblino or on one of his friends (new characters invented for the app, such as a pirate's parrot).

Compare this with developing using a computer: if you already know how to develop an application in C#, in Python, or in JavaScript, you have a great place to start. But if you don't know, you first have to evaluate and choose a language and then work out how to write it, get it going, and make it start automatically. Any one of these tasks may be, strictly speaking, *easier* than any of the more opaque interactions with an odd-looking circuit board, but the freedom of choice adds its own complexities.

Another option is to marry the capabilities of a microcontroller to connect to low-level components such as dials, LEDs, and motors while running the hard processing on a computer or phone. A kit such as an Arduino easily connects to a computer via USB, and you can speak to it via the serial port in a standard way in any programming language.

Some phones also have this capability. However, because phones, like an Arduino, are "devices", in theory they can't act as the computer "host" to control the Arduino. (The side of the USB connection usually in charge of things.) The interesting hack used by the Android development kit (ADK), for example, is for the Arduino to have a USB host shield—that is, it pretends to be the computer end of the connection and so in theory controls the phone. In reality, the phone does the complicated processing and communication with the Internet and so on.

As always, there is no single "right answer" but a set of trade-offs. Don't let this put you off starting a prototype, though. There are really no "wrong answers" either for that; the prototype is something that will get you started, and the experience of making it will teach you much more about the final best platform for your device than any book, even this one, can.

# PROTOTYPES AND PRODUCTION

Although ease of prototyping is a major factor, perhaps *the* biggest obstacle to getting a project started—scaling up to building more than one device, perhaps many thousands of them—brings a whole new set of challenges and questions.

## CHANGING EMBEDDED PLATFORM

When you scale up, you may well have to think about moving to a different platform, for cost or size reasons. If you've started with a free-form, powerful programming platform, you may find that porting the code to a more restricted, cheaper, and smaller device will bring many challenges. This issue is something to be aware of. If the first prototype you built on a PC, iPhone,

BeagleBone, or whatever has helped you get investment or collaborators, you may be well placed to go about replicating that compelling functionality on your final target.

Of course, if you've used a constrained platform in prototyping, you may find that you have to make choices and limitations in your code. Dynamic memory allocation on the 2K that the Arduino provides may not be especially efficient, so how should that make you think about using strings or complex data structures? If you port to a more powerful platform, you may be able to rewrite your code in a more modern, high-level way or simply take advantage of faster processor speed and more RAM. But will the new platform have the same I/O capabilities? And you have to consider the ramping-up time to learn new technologies and languages.

In practice, you will often find that you don't need to change platforms. Instead, you might look at, for example, replacing an Arduino prototyping microcontroller with an AVR chip (the same chip that powers the Arduino) and just those components that you actually need, connected on a custom PCB. We look at this issue in much more detail in Chapter 10.

## PHYSICAL PROTOTYPES AND MASS PERSONALISATION

Chances are that the production techniques that you use for the physical side of your device won't translate directly to mass production. However, while the technique might change—injection moulding in place of 3D printing, for example—in most cases, it won't change what is possible.

An aspect that may be of interest is in the way that digital fabrication tools can allow each item to be slightly different, letting you personalise each device in some way. There are challenges in scaling this to production, as you will need to keep producing the changeable parts in quantities of one, but *mass personalisation*, as the approach is called, means you can offer something unique with the accompanying potential to charge a premium.

## CLIMBING INTO THE CLOUD

The server software is the easiest component to take from prototype into production. As we saw earlier, it might involve switching from a basic web framework to something more involved (particularly if you need to add user accounts and the like), but you will be able to find an equivalent for whichever language you have chosen. That means most of the business logic will move across with minimal changes.

Beyond that, scaling up in the early days will involve buying a more powerful server. If you are running on a cloud computing platform, such as Amazon Web Services, you can even have the service dynamically expand and contract, as demand dictates.

## Case Study: DoES Liverpool

Some of the reasons to think about production techniques may actually simply be about having a more beautiful device, even for your own use. In later chapters we see some methods that can be used to clad and house an Internet of Things prototype device. But there may well be a progression from unhoused electronics to something in a box made from LEGO or using a repurposed casing to a device in a custom-made box (laser-cut, 3D printed) or even one machined commercially.

In our office and makespace, DoES Liverpool, the central heating system has been hooked up to the Internet. YAHMS, as the system is named, consists of a collection of sensors to measure temperature in the office and outside, an actuator to turn the heating on or off, and some server software to manage timer control and provide a web-based interface to the system. Like many non-Internet-connected heating systems, there is a timer-based programme which ensures a basic level of comfort automatically. However, users can log on to the YAHMS website to find out what the temperature is and decide to turn the heating on or off to override the programme. As it is web-based, it works equally well if you're sitting in the office at the time or if you're at home getting ready to head into work on a cold winter Saturday.

The temperature sensors have been left as unhoused Arduino boards. They are managed only by John, whose project it is. However, the cabling to the boiler itself is neatly installed, and the electronics are hidden away. The actual interface that people use every day, however, is well styled with a minimal interface and works equally well on a desktop browser or a smartphone. Mimicking the hobbyist-consumer scale that we saw for hardware, the software also has a similar scale: some tasks don't currently expose their functionality through the UI and require manual tweaking with database commands. Clearly, the whole system isn't currently suitable for mass sale, but it is serviceable, it fits into the ethos of tinkering and "interesting tech" in the location where it is installed, and the most common tasks that it carries out are well packaged.

Another example from DoES is the DoorBot. It originally consisted of a networked PC with a flat-screen monitor facing out towards the corridor through a conveniently located window. The DoorBot works as a kiosk device, showing webcam views of the office, a list of upcoming events (regularly pulled from Google Calendar), and a welcome message to any expected guests. Currently, its only input device is an RFID reader. Our members can register their RFID cards (Oyster, Walrus, DoES membership card, and so on). Finally, this device is also connected to speakers, so it can play a personalised tune or message when members check in or out. Developing this device was as simple as running software on a computer ever is: the trickiest cases are things such as turning the screen off and on after office hours and coping with losing or regaining power and network. Given how close the functionality is to that of a PC, it might seem crazy to think of

any other solution. However, if we had to scale up—to cover more doors or to sell the idea to other companies—we suddenly have new trade-offs.

Just sticking a tower PC somewhere near the door may not be ideal for every office. A computer that fits neatly with an integrated screen might work, such as an iMac, a laptop, or a tablet. But these devices are much more expensive than the original commodity PC (effectively "free" when it was a one-off because it was lying around with nothing else to do). A small embedded computer, such as a Raspberry Pi, might be ideal because it costs relatively little, runs Linux, and has HDMI output.

Our co-workers, John and Ben, did eventually port the DoorBot to the Raspberry Pi, and this was a great learning experience with that platform. However, it was still an investment in time even though it was theoretically a simple change, as the Pi has similar capabilities to a Linux PC. The primary driver for investing the time to do this came from our expansion to a second and now third room—requiring three devices made finding a smaller, cheaper, more polished solution a requirement. We look at this process in greater detail in the next chapter.

For projects that aren't as clearly suited to being a computer and that do need to interface to electronics components, you are more likely to consider the continuum from basic microcontrollers (Arduino and others) to more powerful ones, such as the BeagleBone.

If you can already run on a smaller, cheaper platform, then if you suddenly have to start providing large numbers of devices, your costs will already be much more reasonable. You've already proved that your concept can run on a cheap, small, limited board. Of course, if your device is beating against the upper limits of the hardware capabilities, you may find that adding any new feature means you suddenly have to upgrade to a more powerful version of your chip or even change to a different platform. If you already have devices on the market, this would affect whether you can easily upgrade an existing device to a new version of firmware, which could be a disadvantage.

If you started with a more complex board, you may have reaped benefits from being able to be freer with dynamic memory allocation, libraries with nicely wrapped APIs, or other handy assumptions. But these assumptions may turn out to be painful if you do have to port the code to a more constrained device: you may be looking at a complete rewrite. Of course, rewriting is not necessarily a problem but has to be factored into your development costs.

## OPEN SOURCE VERSUS CLOSED SOURCE

If you're so minded, you could spend a lifetime arguing about the definitions of "closed" and "open" source, and some people have, in fact, made a career out of it. Broadly, we're looking at two issues:

- Your assertion, as the creator, of your Intellectual Property rights
- Your users' rights to freely tinker with your creation

We imagine many of this book's readers will be creative in some sense, perhaps tinkerers, inventors, programmers, or designers. As a creative person, you may be torn between your own desire to learn how things work and modify and re-use them and the worry that if *other people* were to use that right on your own design/invention/software, you might not get the recognition and earnings that you expect from it.

In fact, this tension between the closed and open approaches is rather interesting, especially when applied to a mix of software and hardware, as we find with Internet of Things devices. While many may already have made up their minds, in one or the other direction, we suggest at least thinking about how you can use both approaches in your project.

## The Open Source Hardware Association

Alongside the emergence of the Internet of Things, a similar, if slightly more advanced, rise of the Maker movement has occurred. This hands-on approach to playing around with technology encourages passionate amateurs and professionals alike to break out the soldering iron and their toolkit and create everything from robots and 3D printers to interactive artworks and games.

A heavy overlap exists between the Maker movement and hackspaces—the network of groups of like-minded individuals helping each other learn new skills and experiment with technology in novel ways. The democratic and open approach from hackspaces, along with that of the open source software community, has carried over into the Maker culture, resulting in a default stance of sharing designs and code.

As the community has grown and matured, this sharing has coalesced into the more formal form of open hardware.

In March 2010, at the Opening Hardware workshop, the participants started to define what open hardware is. Over the following 11 months, both online and at the first Open Hardware Summit, this definition was honed and refined into version 1.0 of the Open Source Hardware (OSHW) Statement of Principles and Definition (`http://freedomdefined.org/OSHW`), which was released in February 2011. Work on the definition is ongoing, and at the time of writing the draft for version 1.1 is a work in progress.

This definition is now accompanied by an Open Source Hardware logo, which designers can use on their products to indicate that the source files are available for learning, re-using, and extending into new products.

The Open Hardware Summit also established itself as a regular annual event, bringing together a huge number of hackers, makers, developers, and designers to discuss and celebrate open hardware each September.

Finally, in June 2012, the Open Source Hardware Association (`http://www.oshwa.org`) was formally incorporated. As we write, the organisation is still finding its feet and working out exactly how to undertake its aims of supporting the Open Hardware Summit and educating the wider public on all matters pertaining to open source hardware. Products which conform to the aims of open source hardware can use the logo.


The OSHWA logo.
`http://oshwlogo.com`

## WHY CLOSED?

Asserting Intellectual Property rights is often the default approach, especially for larger companies. If you declared copyright on some source code or a design, someone who wants to market the same project cannot do so by simply reading your instructions and following them. That person would have to instead reverse-engineer the functionality of the hardware and software. In addition, simply copying the design slavishly would also

infringe copyright. You might also be able to protect distinctive elements of the visual design with trademarks and of the software and hardware with patents.

Although getting good legal information on what to protect and how best to enforce those rights is hard and time-consuming, larger companies may well be geared up to take this route. If you are developing an Internet of Things device in such a context, working within the culture of the company may simply be easier, unless you are willing to try to persuade your management, marketing, and legal teams that they should try something different.

If you're working on your own or in a small company, you might simply trademark your distinctive brand and rely on copyright to protect everything else. Note that starting a project as closed source doesn't prevent you from later releasing it as open source (whereas after you've licensed something as open source, you can't simply revoke that licence).

You may have a strong emotional feeling about your Intellectual Property rights: especially if your creativity is what keeps you and your loved ones fed, this is entirely understandable. But it's worth bearing in mind that, as always, there is a trade-off between how much the rights actually help towards this important goal and what the benefits of being more open are.

## WHY OPEN?

In the open source model, you release the sources that you use to create the project to the whole world. You might publish the software code to GitHub (`http://github.com`), the electronic schematics using Fritzing (`http://fritzing.org`) or SolderPad (`http://solderpad.com`), and the design of the housing/shell to Thingiverse (`http://www.thingiverse.com`).

If you're not used to this practice, it might seem crazy: why would you give away something that you care about, that you're working hard to accomplish? There are several reasons to give away your work:

- You may gain positive comments from people who liked it.
- It acts as a public showcase of your work, which may affect your reputation and lead to new opportunities.
- People who used your work may suggest or implement features or fix bugs.
- By generating early interest in your project, you may get support and mindshare of a quality that it would be hard to pay for.

Of course, this is also a gift *economy*: you can use other people's free and open source contributions within your own project. Forums and chat channels exist all over the Internet, with people more or less freely discussing their projects because doing so helps with one or more of the benefits mentioned here.

If you're simply "scratching an itch" with a project, releasing it as open source may be the best thing you could do with it. A few words of encouragement from someone who liked your design and your blog post about it may be invaluable to get you moving when you have a tricky moment on it. A bug fix from someone who tried using your code in a way you had never thought of may save you hours of unpleasant debugging later. And if you're very lucky, you might become known as "that bubble machine guy" or get invited to conferences to talk about your LED circuit.

If you have a serious work project, you may *still* find that open source is the right decision, at least for some of your work.

## Disadvantages of Open Source

The obvious disadvantage of open source—"but people will steal my idea!"—may, in fact, be less of a problem than you might think. In general, if you talk to people about an idea, it's hard enough to get them to listen because they are waiting to tell you about *their* great idea (the selfish cads). If people do use your open source contribution, they will most likely be using it in a way that interests *them*. The universe of ideas is still, fortunately, very large.

However, deciding to release as open source may take more resources. As the saying goes: the shoemaker's children go barefoot. If you're designing for other people, you have to make something of a high standard, but for yourself, you often might be tempted to cut corners. When you have a working prototype, this should be a moment of celebration. Then having to go back and fix everything so that you can release it in a form that doesn't make you ashamed will take time and resources.

Of course, the right way to handle this process would be to start pushing everything to an open repository immediately and develop in public. This is much more the "open source way". It may take some time to get used to but may work for you.

After you release something as open source, you may still have a perceived duty to maintain and support it, or at least to answer questions about it via

email, forums, and chatrooms. Although you may not have *paying* customers, your users are a community that you may want to maintain. It is true that, if you have volunteered your work and time, you are entirely responsible for choosing to limit that whenever you want. But abandoning something before you've built up a community around it to pass the reins to cannot be classed as a successful open source project.

## Being a Good Citizen

The idea that there is a "true way" to do open source is worth thinking about. There is in some way a cachet to "doing open source" that may be worth having. Developers may be attracted to your project on that basis. If you're courting this goodwill, it's important to make sure that you do deserve it. If you say you have an open platform, releasing only a few libraries, months afterwards, with no documentation or documentation of poor quality could be considered rude. Also, your open source work should make some attempt to play with other open platforms. Making assumptions that lock in the project to a device you control, for example, would be fine for a driver library but isn't great for an allegedly open project.

In some ways, being a good citizen is a consideration to counterbalance the advantages of the gift economy idea. But, of course, it is natural that any economy has its rules of citizenship!

## Open Source as a Competitive Advantage

Although you might be tempted to be very misty-eyed about open source as a community of good citizens and a gift economy, it's important to understand the possibility of using it to competitive advantage.

First, *using* open source work is often a no-risk way of getting software that has been tested, improved, and debugged by many eyes. As long as it isn't licensed with an extreme viral licence (such as the AGPL), you really have no reason not to use such work, even in a closed source project. Sure, you could build your own microcontroller from parts and write your own library to control servo motors, your own HTTP stack, and a web framework. Or you could use an Arduino, the Arduino servo libraries and Ethernet stack, and Ruby on Rails, for example. Commercial equivalents may be available for all these examples, but then you have to factor in the cost and rely on a single company's support forums instead of all the information available on the Internet.

Second, using open source aggressively gives your product the chance to gain mindshare. In this book we talk a lot about the Arduino—as you have seen in this chapter; one could easily argue that it isn't the most powerful

platform ever and will surely be improved. It scores many points on grounds of cost but even more so on mindshare. The design is open; therefore, many other companies have produced clones of the board or components such as shields that are compatible with it. This has led to amusing things such as the Arduino header layout "bug" (`http://forum.arduino.cc/index.php/topic,22737.0.html#subject_171839`), which is the result of a design mistake that has nevertheless been replicated by other manufacturers to target the same community.

If an open source project is good enough and gets word out quickly and appealingly, it can much more easily gain the goodwill and enthusiasm to become a platform. The "geek" community often choose a product because, rather than being a commercial "black box", it, for example, exposes a Linux shell or can communicate using an open protocol such as XML. This community can be your biggest ally.

## Open Source as a Strategic Weapon

One step further in the idea of open source used aggressively is the idea of businesses using open source strategically to further their interests (and undermine their competitors).

In "Commoditizing your complements" (`http://www.joelonsoftware.com/articles/StrategyLetterV.html`), software entrepreneur Joel Spolsky argues that many companies that invest heavily in open source projects are doing just that. In economics, the concept of *complements* defines products and services that are bought in conjunction with your product—for example, DVDs and DVD players.

If the price of one of those goods goes down, then demand for both goods is likely to rise. Companies can therefore use improvements in open source versions of complementary products to increase demand for their products. If you manufacture microcontrollers, for example, then improving the open source software frameworks that run on the microcontrollers can help you sell more chips.

Simon Wardley, a thought leader in the field of cloud computing, writes

> *For many, the words open source [conjure] up concepts of hippy idealism where geeks in a spirit of free love give away their work to others for nothing. For many, it's about as anti-capitalist as you can get. Those many are as gullible as the citizens of Ancient Troy.*
>
> *—http://blog.gardeviance.org/2012/04/be-wary-of-geeks-bearing-gifts.html*

While open sourcing your core business would be risky indeed, trying to standardise things that you use but which are core to *your competitor*'s business may, in fact, help to undermine that competitor. So Google releasing Android as open source could undermine Apple's iOS platform. Facebook releasing Open Compute, to help efficiently maintain large data centres, undermines Google's competitive advantage.

Facebook clearly needs efficient data centres. So to open source its code gives the company the opportunity to gain contributions from many clever open source programmers. But it gives nothing away about Facebook's core algorithms in social graphing.

This dynamic is fascinating with the Internet of Things because several components in different spaces interact to form the final product: the physical design, the electronic components, the microcontroller, the exchange with the Internet, and the back-end APIs and applications. This is one reason why many people are trying to become leaders in the middleware layers, such as Xively (free for developers, but not currently open source, though many non-core features are open).

While you are prototyping, these considerations are secondary, but being aware of these issues is worthwhile so that you understand the risks and opportunities involved.

## MIXING OPEN AND CLOSED SOURCE

We've discussed open sourcing many of your libraries and keeping your core business closed. While many businesses can exist as purely one or the other, you shouldn't discount having both coexist. As long as you don't make unfounded assertions about how much you use open software, it's still possible to be a "good citizen" who contributes back to some projects whether by contributing work or simply by helping others in forums while also gaining many of the advantages of open source.

While both of us tend to be keen on the idea of open source, it's also true that not all our work is open source. We have undertaken some for commercial clients who wanted to retain IP. Some of the work was simply not polished enough to be worth the extra effort to make into a viable open release.

Adrian's project Bubblino has a mix of licences:

- Arduino code is open source.
- Schematics are available but not especially well advertised.
- Server code is closed source.

The server code was partly kept closed source because some details on the configuration of the Internet of Things device were possibly part of the commercial advantage.

### CLOSED SOURCE FOR MASS MARKET PROJECTS

One edge case for preferring closed source when choosing a licence may be when you can realistically expect that a project might be not just successful but *huge*, that is, a mass market commodity. While "the community" of open source users is a great ally when you are growing a platform by word of mouth, if you could get an existing supply and distribution chain on your side, the advantage of being first to market and doing so cheaper may well be the most important thing.

Let's consider Nest, an intelligent thermostat: the area of smart energy metering and control is one in which many people are experimenting. The moment that an international power company chooses to roll out power monitors to all its customers, such a project would become instantaneously mass market. This would make it a very tempting proposition to copy, if you are a highly skilled, highly geared-up manufacturer in China, for example. If you also have the schematics and full source code, you can even skip the investment required to reverse-engineer the product.

The costs and effort required in moving to mass scale show how, for a physical device, the importance of supply chain can affect other considerations. In 2001, Paul Graham spoke compellingly about how the choice of programming language (in his case, Lisp) could leave competitors in the dirt because all of his competitors chose alternative languages with much slower speed of development (`www.paulgraham.com/avg.html`). Of course, the key factor wasn't so much about development platform as *time to market* versus your competitor's time to market. The tension between open and closed source informs this as well.

## TAPPING INTO THE COMMUNITY

We talked about the "community" in the previous section, but it would be disingenuous to pretend that this is *exclusively* a feature of open source projects.

While thinking about which platform you want to build for, having a community to tap into may be vital or at least useful. Again, this is a major reason for our current support of the Arduino platform. If you have a problem with a component or a library, or a question about how to do something (for example, controlling a servo motor with a potentiometer

dial), you could simply do a Google search on the words "arduino servo potentiometer" and find a YouTube video, a blog post, or some code.

Many other cute platforms, such as the Chumby Hacker Board, do have communities of aficionados, but perhaps smaller ones. If you are doing something more obscure or need more detailed technical assistance, finding someone who has already done exactly that thing may be difficult.

Mindshare may be important as you scale up, too—for example, if you want confidence that you can hire people with skills in the platform you've chosen. This issue may be less important for a small, focused team which has a lot of expertise in a new or obscure platform but still may be a consideration.

When you are an inexperienced maker, using a platform in which other people can mentor you is invaluable. If you have a local meeting for makers, such as Maker Night Liverpool, or equivalents in hackspaces around the world, you will very often find someone who is willing to take you through the basics in Arduino or another similar system. Perhaps that person is an expert on it or has simply gone through the basics (getting an LED flashing or playing "Mary had a little lamb" with a piezo speaker) at the last meeting. These meetings can be invaluable for both student and mentor.

Local meetings are also a great way to discuss your own project and learn about others. While to discuss your project is in some way being "open" about it, you are at all times in control of how much you say and whom you say it to. If you're not already open source minded, this approach can be much less intimidating than releasing your clever idea to the whole Internet at once.

The perceived danger of sharing an idea or an implementation or a question with other people is looking like an idiot in public. While many parts of many Internet communities are much more sympathetic to this fear than one might expect, the mask of anonymity on the Internet can seem to permit people to be less supportive or simply more rude than you might hope for. In general, face-to-face meetings at a hackspace may well be a friendlier and more supportive way to dip your toes into the idea of a "community" of Internet of Things makers.

One reason to be in touch with a (local or Internet) community of makers is that we are, in interaction designer and BERG hardware engineer Andy Huntington's words, at the stage of the "Geocities of things"—that is, at the frontier of the Internet of Things, just as Geocities was at the frontier of making websites and blogging. Sure, the design of some of the things may be clunky, the things might be pointless, and a lot of people may simply be

doing something that they saw someone else do before. But from this outpouring of creativity will come the next generation of successful businesses and projects that actually change the world. This is a fascinating time to be getting involved in the Internet of Things.

# SUMMARY

You now have a decent grounding in the wider issues around building your first Internet of Things prototype.

Prototyping is inherently a matter of balancing trade-offs between building something that allows you to learn more about the project you are looking to build and keeping an eye on how things scale up should your experiments prove you right.

In the next chapters we take each of the three main components of an Internet of Things device in turn: the embedded computing and electronics; the physical *Thing* itself; and the Internet service to which it talks. We look in more detail at how you would go about prototyping that aspect of your device.

You will be able to frame each element with concepts from this chapter to guide your choices—whether that is deciding to use Ruby on Rails for the server software due to its open source licence; plumping for a Raspberry Pi development board because you're an experienced Python programmer; or downloading a 3D design for a key component from Thingiverse to save having to design one from scratch.